A Concurrent Implementation of the Cascade-Correlation Algorithm, Using the Time Warp Operating System

Paul Springer
Jet Propulsion Laboratory
California Institute of Technology

Abstract

This paper discusses the method in which the Cascade-Correlation algorithm was parallelized in such a way that it could be run using the Time Warp Operating System (TWOS). TWOS is a special purpose operating system designed to run parallel discrete event simulations with maximum efficiency on parallel or distributed computers. Timings are also given to indicate how efficient the parallelization process was for a particular benchmark.

1. Introduction

Cascade-correlation has been shown to have important advantages over back propagation as a learning algorithm for neural networks. Because cascade-correlation builds hidden units into the neural network as it goes, the neural network designer is relieved of the task of having to guess at the best configuration of hidden units for a particular problem. Studies have shown cascade-correlation to be a faster algorithm and better able to converge. [1;2] Whitley and Karunanithi [3] write that cascade-correlation also shows very good generalization characteristics.

Compared to the research work which has gone into parallelizing back propagation, not much has been published regarding the parallelization of the cascade-correlation algorithm. Given the advantages of cascade-correlation, research in this area. would seem to be worthwhile. This paper describes one such effort.

Section 2 of the paper describes some of the fundamentals of cascade-correlation. Section 3 describes the Time Warp Operating System. Details of the implementation appear in section 4, and section 5 describes the parallelism in this implementation. Section 6 concludes with some performance measurements that were done on a BBN GP1 000 parallel computer.

2. Cascade-Correlation

Cascade-correlation is a neural network algorithm which not only trains a neural network, but also dynamically builds the network architecture. The number of output units is specified initially, and there are no hidden units at the outset. In the course of training, hidden units are added to the network layer by layer, with a single hidden unit in each layer.

The cascade-correlation algorithm cycles through two phases. In the first phase, the output units are trained and their weights adjusted until no further progress is made. Fahlman's quickprop algorithm [4] is used to adjust the weights in this

phase.

In the second phase, a number of candidate units are set up to receive inputs from the network's external inputs as well as inputs from the hidden units which have previously been added to the network. Each candidate unit is trained to maximize C, the correlation of its output with the error signal previously existing in the network. The formula is

$$C = \sum_o \left| \sum_p (y_p - \bar{y})(e_{op} - \bar{e}_o) \right| \qquad (1)$$

where y_p represents the candidate's output for pattern p, e_{op} is the residual error observed in the active network for output unit o , on pattern p. *The* average of y over all patterns is represented by y, and e_o represents the average of e. over all patterns p.

C is maximized by gradient ascent using the partial derivative of C with respect to each of the candidate unit's incoming weights w_i:

$$\partial C / \partial w_i = \sum_{p,o} s_o (e_{op} - \bar{e}_o) f'_p I_{i,p} \qquad (2)$$

where so is the sign of the correlation between the candidate's value and output o, f'_p is the derivative of the candidate unit's activation function for pattern p with respect to the sum of its nputs, and $I_{i,p}$ is the input the candidate unit receives from unit i for pattern p.

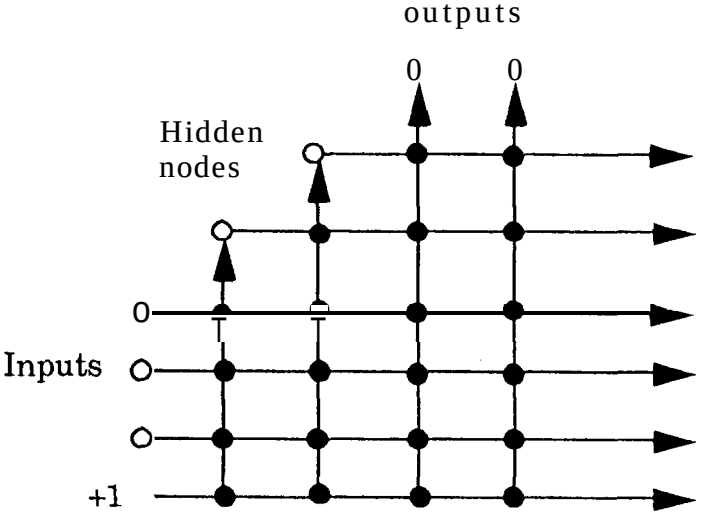


Figure 1

At the end of this second phase, the candidate unit with the best correlation is chosen to be the next hidden unit to be added to the network (see Figure 1). Its weights are then frozen, and the first phase is started again by retraining the output units using the inputs available previously, as well as the signal from the new hidden unit. A more detailed explanation of cascade-correlation appears in [1].

3. **The Time Warp Operating System**

When a program is decomposed into individual components or objects which run in parallel, there generally must be some way to synchronize these objects, Optimistic synchronization has shown itself to be one of the best ways to accomplish this. Optimistic synchronization permits an object to execute even though the object in question may not yet have all the inputs it needs. The synchronization method takes care of any errors that may arise due to the premature execution of an object.

In contrast, conservative methods of synchronization may not be able to extract as much parallelism from a discrete event simulation as optimistic methods can. A simulation using a conservative method will block if there is even a remote chance of a causality error occurring. An optimistic method would not block under this circumstance, but would proceed. If no causality error resulted, the optimistic method wins. If an event was run out of order, this situation is detected and the optimistic method restores the state of the simulation to what it was previously, and runs the events in the proper order. For a good overview which contrasts these techniques, see [51.

The Time Warp Operating System (TWOS) uses the best known method of optimistic synchronization, Time Warp. Time Warp is based on the concept of virtual time, When a message arrives for an object, and the message is to be received at a time earlier than the object's current simulation time, the obj ect's current simulation time is rolled back to that earlier time, and execution is reinitiated. See [6] for a more detailed explanation of Time Warp.

This implementation of cascade-correlation was written in such a way that it could be run in parallel as a TWOS application. It was written in the form of a discrete event simulation, with each epoch being run at a discrete point in simulation time.

Time Warp requires that objects communicate with each other by means of messages which contain time tags to indicate at what simulation time the receiving object should begin execution of an event. An object begins executing at a specific simulation time only when it has received a message which is tagged with that specific receive time.

A key characteristic of Time Warp is that an object may execute at a simulation time when not all messages for that time have yet been received by that object. If an object receives a message with a receive time earlier than the current simulation time at which the object is executing, the object is rolled back to the earlier time, The rolled-back object will then re-execute its work starting from the earlier time.

Suppose an object has just completed processing a message which had a receive time of 1300. Now suppose a message arrives with a receive time of 1200. Any modifications which the object made to its environment between time 1200 and time 1300 must be reversed. In particular, previous values of variables must also be restored. TWOS accomplishes this by forcing the user to group any variables which persist from one event into another into a special structure called a state. After each event, a copy of the object's state is saved, so that the object can be rolled back to that state, if necessary.

Two other types of variables are supported by ***TWOS:*** stack variables and read-only global variables. As the name indicates, stack variables are allocated space on the object's stack, and disappear when the object finishes executing the current event. Global variables, which could be shared between objects, present something of a problem, however. TWOS saves only previous versions of an object's state, and does not attempt to save previous versions of global variables. Therefore, general purpose global variables are not permitted in a TWOS application. However, TWOS does permit a special kind of global variable called a *read-only global.* These global variables can be initialized only once, before the body of the simulation begins. Once initialized, they can never be modified.

TWOS allows the use of dynamically allocated memory, but not by means of a standard system call. Instead an application must use a special TWOS call to allocate the memory. This call ties the allocated memory to the state of the object making the memory request. TWOS creates a version of an object's dynamically allocated memory for each event executed by the object, handling this process the same way that it does for the object's states.

4. Implementation Details

There were three main challenges in implementing the cascade correlation algorithm to work with TWOS: (1) making an object based design; (2) setting up each object's state; and (3) setting up the read-only global area.

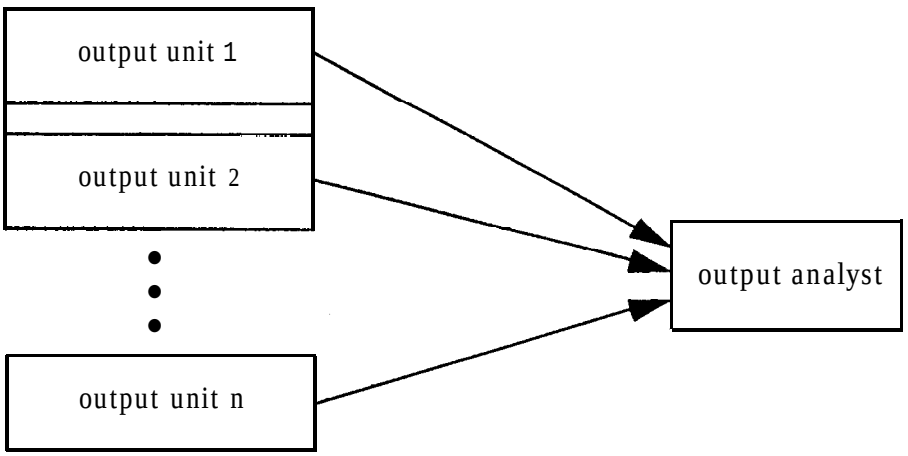
I decided upon four types of objects for the application: an output type, an output analyst type, a candidate type, and a candidate analyst type. All TWOS objects are instantiated at the beginning of a run, by means of a TWOS configuration file which specifies and names each object.

An output type of object is created for each output unit in the network. After initializing itself, an output object makes a pass through the set of training patterns. At the end of each such epoch, the object calculates the error and various other mea surements which are used to adjust the weights for the next epoch, using the quickprop algorithm. The object then sends a message to itself to trigger the next epoch, The error information from each output object is sent to the output analyst object.

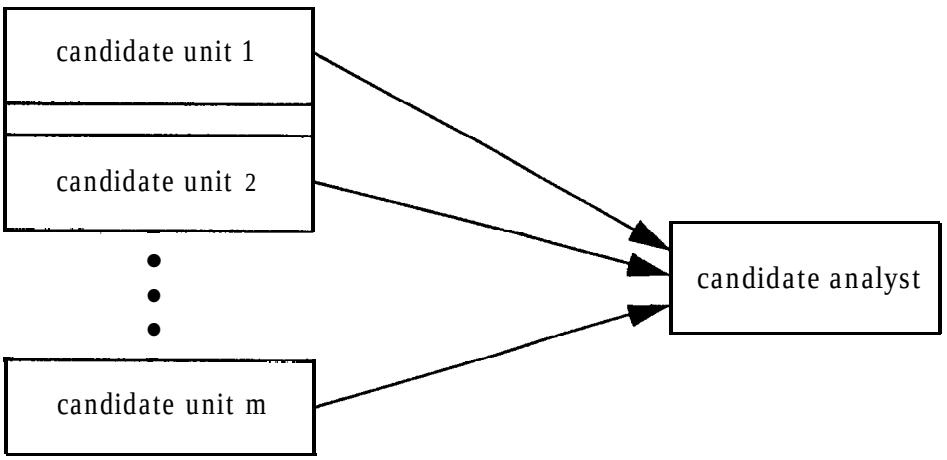
Only a single output analyst object is created. It receives error information from every output object at the end of each epoch. The output analyst determines whether or not the output units are continuing to make progress. If not, then the output,

analyst sends a message to each output object which causes that output object to stop cycling through the test patterns. Receipt of this message also causes each output object to send error information to each candidate object in the candidate pool.

When the candidate objects receive error information from the output objects, the simulation leaves the output unit training phase and enters the candidate unit training phase of execution (see Figure 2). During this new phase, the output units are idle and the candidate units are active, As each candidate object passes through an epoch, it measures the correlation of its output with the error information passed to it, and then adjusts its weights to maximize that correlation. At the end of an epoch, each candidate object sends a message to itself in order to trigger the next epoch. The candidate object also sends a message to the candidate analyst to report its correlation value.



Output unit training phase



Candidate unit training phase

Figure 2

As with the output analyst, only a single candidate analyst object is created. All candidate objects report their correlation values to the candidate analyst at the end of each candidate epoch. When the candidate analyst determines that little or no additional progress is being made with each epoch, it sends messages to all the candidate units to stop cycling. The candidate analyst also selects which candidate has the best score, and sends a "win" message to that candidate.

When a candidate object receives a "win" message, it makes one final pass through the training patterns. Using its current weights, it produces a "value" array which contains the value of the candidate unit's output corresponding to each training pattern. In the current implementation, the candidate unit is not actually added to the network as a hidden unit. Instead, each of the training patterns is effectively enlarged by the appropriate entry from this "value" array. It is possible to do this because the input weights belonging to hidden units are never modified in the cascade-correlation algorithm. This technique uses more memory in each of the candidate and output objects, but allows a greater degree of parallelism.

The "value" array from the selected candidate object is passed to all output objects and all candidate objects. The receipt of this message signals the output units to begin retraining. The program then exits the candidate unit training phase, and reenters the output unit training phase. The entire program terminates when the output analyst determines that the error from the output units is sufficiently small.

Each of the objects has a state associated with it which contains variables used in the simulation. These variables contain status information (such as the current epoch number) and parameter information (such as the epsilon value used for quickprop). Additionally, the output and candidate objects have dynamically allocated memory segments which are used to store weight information, correlation values, error values, etc.

All the objects share a read-only global data area. The program sets up this area in the initialization stage of the simulation. The program reads a data file which contains the training patterns as well as information on the number of inputs and outputs and certain parameter information. The contents of the data file are used to initialize the read-only global data area.

As mentioned earlier, the read-only global data area can not be modified once the body of the simulation begins. Also mentioned previously was the statement that each of the training patterns is enlarged in place of adding a new hidden unit. Because the read-only area can not be modified, the enlarged portion of each training pattern resides in a dynamically allocated segment of memory. One such segment is associated with each output object and each candidate object.

5. **Parallelism**

The degree of parallelism in this application changes between the output unit training phase and the candidate unit training phase, Assuming the output analyst object and each of the output objects are on different nodes, those objects can run in

parallel. Because optimistic synchronization is used, each of the output objects can forge ahead at its own rate, without waiting for the output analyst to decide whether to switch over to the next phase. So there is parallelism not only among the output objects, but also between the output objects and the output analyst object. Once the output analyst decides it is time to switch phases, it sends a message to each of the output objects. The message causes each output object to roll back to the proper simulation time and stop cycling.

Similarly, the candidate objects can also execute in parallel. Assuming all are on. separate nodes, the candidate analyst is able to determine how far in the cycle to proceed without holding up the progress of the candidate objects themselves. When the candidate analyst determines that it is time to switch phases, it sends messages to all the candidate objects which cause them to roll back to the proper simulation time.

The maximum possible speedup is different for each of the two phases of the program. While the program is in the output unit training phase, the speedup is limited by the number of output objects running in parallel, This in turn is a function of the total number of output objects, and the number of output objects per node. Sequential execution time for this phase of the program can be described by the following formula:

$$T_1 = \sum_{i=1..c} (Ai + \sum_{j=1..n} K_{ij}) \qquad (3)$$

Here T_1 is the execution time for the output training phase, c the number of epochs in this phase, **Ai** the execution time required by the output analyst object for each epoch i, and $K_{,j}$ is the execution time used by output object j during epoch i. The total number of output objects is represented by n.

To simplify speedup analysis, we can assume that each output unit has the same amount of work to do in each epoch, that is $K_{ix} = K_{i1}$ for all x. We can also assume that the execution time of the output analyst is less than the execution time of an output object, unless the number of output units were to grow very large.

By having each output unit and the output analyst executing on its own node, the smallest possible value for elapsed time would be the amount of time it takes an object to execute, namely E_1.

$$E_1 = \sum_i K_{i1} \qquad (4)$$

Dividing (3) by (4) gives the maximum possible speedup for this phase, call it S_1:

$$S_1 = \sum_1 (A_i + nK_{i1}) \; / \; \sum_1 K_{i1} = n + \sum_i A_i \; / \; \sum_i K_{i1} \qquad (5)$$

Assuming $Ai < Ki$, as mentioned above, our maximum possible speedup with this

configuration is between n and n+l.

When the program is in the candidate unit training phase, the speedup is limited by the number of candidate units running in parallel. Again, this is a function of the total number of candidate objects as well as the number of candidate objects per node, Speedup analysis is similar to the speedup analysis for the other phase. Se quential execution time for the candidate unit training phase of the program can be described by the following formula:

$$T_2 = \sum_{i=1..d} (Bi + \sum_{j=1..m} L_{ij}) \qquad (6)$$

Here T_2 is the execution time for the 'candidate unit training phase, d the number of epochs in this phase, B_i the execution time required by the candidate analyst object for each epoch i, and L_{ij} is the execution time used by candidate object j during epoch i. *The* total number of candidate objects in the pool is represented by m.

As above, we can assume that each candidate unit has the same amount of work to do in each epoch, that is $L_{ix} = L_{i1}$ for all x. We can also assume that the execution time of the candidate analyst is less than the execution time of a candidate object, unless the number of candidate units were to grow very large.

By having each candidate unit and the candidate analyst executing on its own node, the smallest possible value for elapsed time would be the amount of time it takes a candidate to execute, namely E_2.

$$E_2 = \sum_i L_{i1} \qquad (7)$$

Dividing (6) by (7) gives the maximum possible speedup for the candidate unit training phase. Calling this speed up S_2 we have:

$$S_2 = \sum_i (B_i + mL_{i1}) / \sum_i L_{i1} = m + \sum_i B_i / \sum_i L_{i1} \qquad (8)$$

Assuming $B_i < L_i$, as mentioned above, our maximum possible speedup during this phase with this configuration is between m and m+l.

Disregarding overhead, the total sequential execution time for the program would be $T_1 + T_2$. Calling this sum T, the formula for the maximum possible speedup S' in this configuration is:

$$S = (T_1 / T)\, S_1 + (T_2 / T)\, S_2 \qquad (9)$$

6. Performance Measurements

The first goal in examining performance of this parallel implementation was to determine how much overhead was introduced by rewriting the cascade correlation

program to work in the Time Warp environment. This was measured by comparing the execution time of a standard C implementation against the execution time of a sequential Time Warp implementation. The standard C implementation was written by R. Scott Crowder, III, and is a public domain program available by ftp. The sequential Time Warp version was created by running the Time Warp version of the cascade correlation code in the Time Warp sequential simulator environment.

The sequential simulator executes code written for TWOS on a single processor in the most efficient way possible. Because it runs on a single node, all events can be run in proper time order, and there is no overhead for such activities as rollbacks or state saving. The sequential simulator was specifically written for speedup measurements.

To benchmark the different implementations of cascade-correlation I chose a learning task that had been worked on here at the Jet Propulsion Laboratory, and which involved Space Shuttle sensor failure detection. This particular application has 9 inputs and 9 outputs, and 496 training patterns. The candidate pool contained 8 candidates.

The Crowder program took 2898 seconds to solve the benchmark problem on a Sun 3/60. In the process, it went through 877 output training epochs, and 86 candidate training epochs. The sequential simulator version solved the same problem on the same computer in 1993 seconds, using 503 output training epochs and 55 candidate training epochs. (Even though the parameters used for each program were the same, it proved impossible to force the two programs to reach a solution in the same number of epochs because of the sensitivity of floating point results to inconsequential differences in program code.)

In this benchmark, candidate training epochs took the same amount of time to execute as did output training epochs, because although the benchmark used had 9 output units and 8 candidate units in the candidate pool, this was balanced by the fact that a candidate epoch executes slightly more code than an output epoch, Thus the Crowder program executed a total of 963 epochs in 2898 seconds for an average of 3.01 seconds per epoch, while the figure for the sequential simulator is 3.57 seconds per epoch. Comparing 3.57 with 3.01 we arrive at a figure of 19% for the extra overhead involved in running a Time Warp version of cascade correlation.

Next to be examined was the amount of speedup realized by running the Time Warp version on multiple nodes under *TWOS*. Here, the comparison was made between the program running under the sequential simulator on one node as opposed to running under TWOS on multiple nodes. The hardware used in this case was the BBN Butterfly, model GP1OOO. Because both the sequential simulator and TWOS versions used the same floating point code, elapsed time figures could be used for speedup comparisons. The TWOS version was run on 10 nodes. This was determined to be the most efficient configuration for this particular benchmark because the program cycled between two phases: in one phase nine output objects and one analyst object executed simultaneously, and in the other phase eight candidate objects and one analyst object executed simultaneously.

Onthe Butterfly, theelapsed time for the sequential simulator version of the program was 2263 seconds. Running the program on 10 nodes under TWOS, the elapsed time was 278 seconds. Dividing one time by the other results in a speedup figure of 8.14. This compares well with the maximum theoretical speedup calculated in section 6. In this instance n in equation (5) is 9 and m in equation (8) is 8. From equation (9) we can deduce that the maximum speedup would be in the range of 8 to 10.

Note that in all the preceding time measurements, program setup time (which included time for such activities as program forking and reading training patterns into memory) was not included. These times were not included because the Butterfly computer being used did not parallelize these activities.

Acknowledgments

I would like to thank Sandeep Gulati of the Jet Propulsion Laboratory for motivating me to write this paper, and for his many helpful suggestions.

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] Scott E, Fahlman and Christian Lebiere, "The Cascade-Correlation Learning Architecture", *Advances in Neural Information Processing Systems 2,* edited by D. S. Touretzky, Morgan Kaufmann, 1990.

[2] Marijke F. Augusteijn and Arturo S. Dimalanta, "Feature Detection in Satellite Images Using Neural Network Technology", *1992 Goddard Conference on Space Applications of Artificial Intelligence,* pp. *123-136.*

[31 D. Whitley and N. Karunanithi, "Generalization in Feed Forward Neural Networks", 1991 *International Joint Conference on Neural Networks,* Vol. II, pp. 77-82, July 1991, Seattle, WA.

[41 *S.* E. Fahlman, "Faster-Learning Variations on Back-Propagation: An Empirical Study", *Proceedings of the 1988 Connectionist Models Summer School,* Morgan Kaufmann, 1988.

[51 Fujimoto, Richard M., "Parallel Discrete Event Simulation", *Communications of the ACM,* October, 1990.

[61 David Jefferson, *et. al.,* "Distributed Simulation and the Time Warp Operating System," *Proceedings of the 11th Annual Symposium on Operating Systems Principles, 1987.*